Arrays, Loops & Pointers

CMPT 125
Jan. 11

Lecture 4

Today

- Arrays and loops
- Performance of loops
- Arrays vs pointers

List vs Array

Python `list`

- a sequence of data
- access by `[index]`
- index from `[0]..[len-1]`
- dynamic length
- can mix types

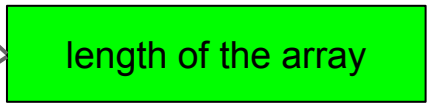
C array

- a sequence of data
- access by `[index]`
- index from `[0]..[len-1]`
- fixed length
- all same type

Array Syntax

```
int main ( ) {
    int labscores[10] = {10,10,9,5,10, 0,10,9,8,10};
}
```

OR:




```
int main ( ) {
    int labscores[10];
    labscores[0] = 10; labscores[1] = 10;
    labscores[2] = 9; labscores[3] = 5;
    labscores[4] = 10; labscores[5] = 0;
    labscores[6] = 10; labscores[7] = 9;
    labscores[8] = 8; labscores[9] = 10;
}
```

Arrays & Iteration

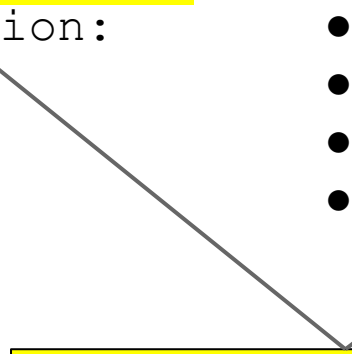
With sequences usually comes iteration.

Python iteration

- `for i in range(n):`
- `while condition:`
- `break`
- `continue`

C iteration

- `for (int i = 0; i < n; i++) { }`
- `while (condition) { }`
- `do { } while (condition);`
- `break;`
- `continue;`



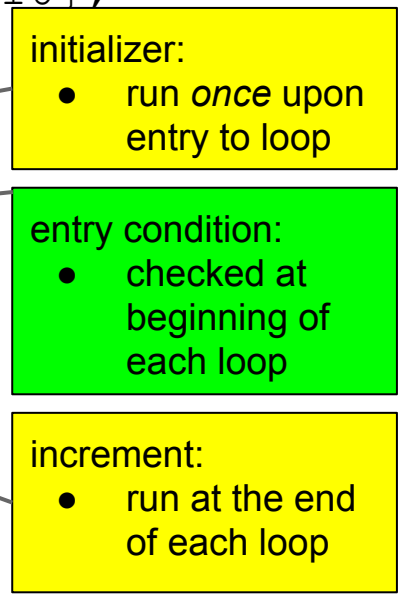
- Main differences in syntax are the `for` loops
- Both are for `0..n-1`

For Loop - Anatomy

```
int main ( ) {
    int labscores[10] = {10,10,9,5,10, 0,10,9,8,10};
    int total = 0;
    float average = 0.0;

    for (int i = 0; i < 10; i++) {
        total = total + labscores[i];
    }
    average = total/10.0;

    printf("Your total score was: %d\n", total);
    printf("Your average score was: %f\n", average);
}
```



Common Errors

```
for (i = 0; i < 10; i++); {
   printf("Score %d: %d", i, labscores[i]);
   total += labscores[i];
}
```

- loop body is a null statement
- intended loop body never executed until `i == 10`

```
for (i = 0; i < 10; i++)
   printf("Score %d: %d", i, labscores[i]);
   total += labscores[i];
```

- loop body doesn't include this statement
- executed once, when `i == 10`

Maximum Style Points: Always use braces, even if loop body is just one statement long.

While Loop

C is virtually the same as Python

Python:

```
def gcd(a, b):
    while b != 0:
        tmp = b
        b = a % b
        a = tmp
    return a
```

C:

```
int gcd(int a, int b) {
    while (b != 0) {
        int tmp = b;
        b = a % b;
        a = tmp;
    }
    return a;
}
```

Conditions behave the same in C as in Python

- 0 treated as False, non-zero treated as True

Running Time of a Loop

```
total = 0;
for (int i = 0; i < N; i++) {
    total += numbers[i];
}
printf("The total is %d\n", total);
```

Loops are a short piece of code that can run for a very long time.

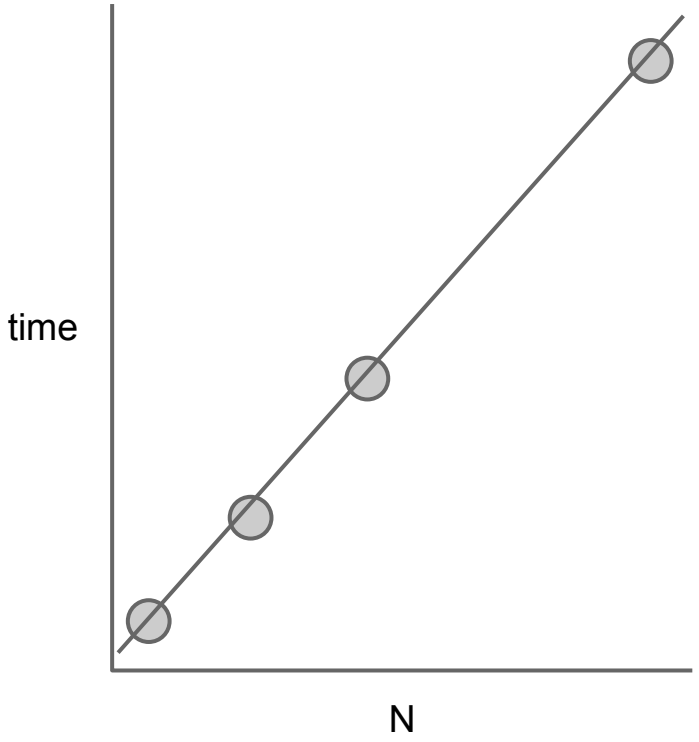
- Can measure time as a function of N.
- As N increases, the running time increases.
- Expect the relationship to be *linear*.

Empirical Measurements

Use a “stopwatch” (the `time` command)

- `time ./a.out`

N	time (in ms)
100000000	252
500000000	1224
1000000000	2394
2000000000	4770



Intuition: As N doubles, the program’s time doubles

Array Bounds

What happens if you access labscores[-1] or labscores[10]?

```
int main ( ) {
    int labscores[10] = {10,10,9,5,10, 0,10,9,8,10};

    for (int i = -1; i <= 10; i++) {
        printf("Your score for lab %d was %d\n", i, labscores[i]
    }
}
```

May cause garbage data *or* crash program (segmentation fault)

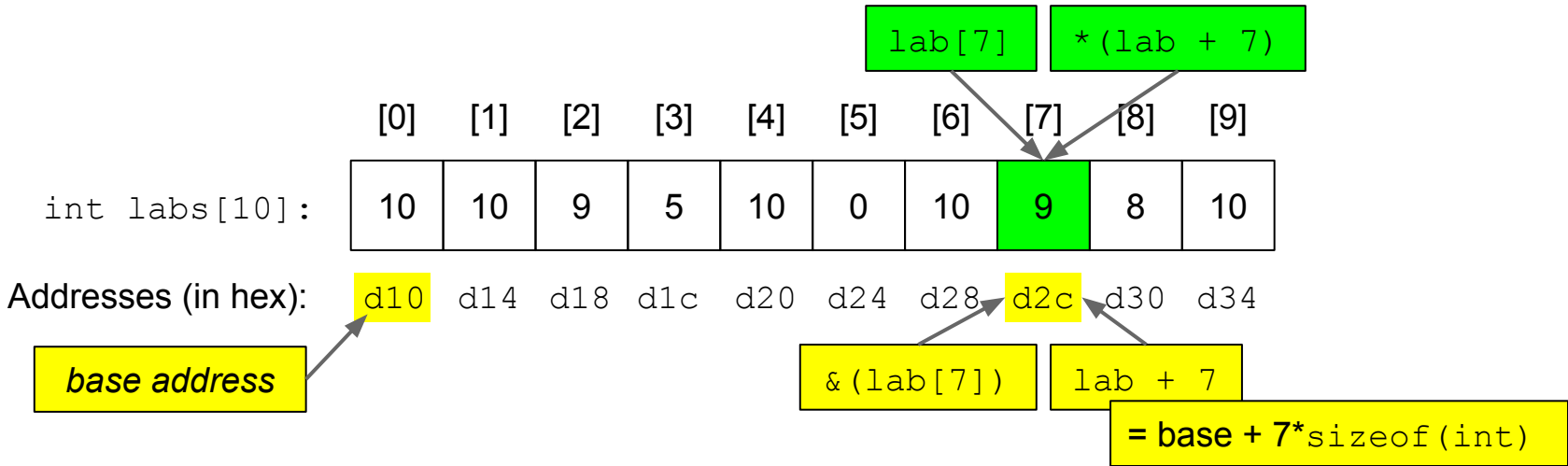
- Python generates a run-time error for labscores[10]

Memory Layout of an Array

```
int main ( ) {
    int labs[10] = {10,10,9,5,10, 0,10,9,8,10};

    for (int i = 0; i < 10; i++) {
        printf("labs[%d] is at 0x%lx\n", i, &labs[i]);
    }
}
```

All array entries are in a *contiguous* space.



Arrays vs Pointers

- The C language treats an array as a pointer
  - points to its base address
  - allows pointer “arithmetic”

```
int main ( ) {
    int labs[10] = {10,10,9,5,10, 0,10,9,8,10};
    int * first = labs;
    int * last = labs + 9;
    for (int * i = first; i <= last; i++) {
        printf("%d is at 0x%lx\n", *i, i);
    }
}
```

`i` iterates through all array elements, initially pointing to the head of the array

`last` points to `labs[9]`. Array bounds are checked every loop.
Alt: `*last = &labs[9]`

i++ means to point to the next element. The pointer itself is increased by 4, the `sizeof(int)`.